2

A RAND NOTE

Prefetching Simulation Objects in a Persistent Simulation Environment

Christopher Burdorf, Stephanie Cammarata

November 1989

RAND

The research described in this report was sponsored by the Defense Advanced Research Projects Agency under RAND's National Defense Research Institute, a federally funded research and development center supported by the Office of the Secretary of Defense, Contract No. MDA903-85-C-0030.

Published by The RAND Corporation
1700 Main Street, P.O. Box 2138, Santa Monica, CA 90406-2138

A RAND NOTE

N-3050-DARPA

Prefetching Simulation Objects in a Persistent Simulation Environment

Christopher Burdorf, Stephanie Cammarata

November 1989

Prepared for the
Defense Advanced Research Projects Agency

RAND

Prefetching Simulation Objects in a Persistent Simulation Environment*

Christopher Burdorf(burdorf@rand.org) **Stephanie Cammarata**(steph@rand.org)
The RAND Corporation
1700 Main Street
Santa Monica, CA 90406-2138

Abstract

We describe a persistent simulation environment (PSE) for object-oriented simulation. PSE is implemented in the Common Lisp Object System (CLOS) with extensions for persistence, object-oriented simulation, and prefetching of objects. Object prefetching optimizes the swapping of persistent objects to and from main memory. It is a type of smart virtual memory system that retrieves objects required by the application program. We discuss prefetching for two different algorithms: Dijkstra's shortest path algorithm and the traveling salesman. Finally, we analyze the use of prefetching by presenting empirical and theoretical results for performance improvements.

1 Introduction

In this paper, we describe our development work on a methodology and prototype architecture for tightly coupling two independent software environments: object-oriented database management systems (OODBMS) and object-oriented simulation languages. The resulting "Persistent Simulation Environment" (PSE) stores input and output simulation data as persistent objects. During simulation processing, PSE's manipulation of persistent entities is transparent to both simulation programmers and users.

The results of this work will contribute to the PRISM (Productivity Improvements in Simulation Modeling) project supported by the Air Force Human Resources Laboratory. The goal of this project is to improve productivity and responsiveness of organizations within the Air Force which provide capability assessments through discrete-event simulation models.

Persistent object systems (POS) which make use of secondary storage will enable object-oriented simulations to be scaled up to efficiently support and maintain many more objects than existing memory-based object-oriented languages. Most POS projects are concerned with seamless integration of simulation language features and traditional data management capabilities, such as transaction management and multi-user access [Atkinson and Buneman 1987, Ford 1988, Khoshafian 1989]. Although these issues are critical to the success of persistent object systems, our efforts specifically focus on one problematic aspect of POS: efficient access of persistent simulation entities. Persistent maintenance of objects on secondary storage has the advantage that objects are no longer tightly coupled to the simulation system; that is, the objects reside in their own repository and can be independently perused before, during, or after a simulation session. However, persistent objects entail disk accesses when the simulation requires objects not resident in the simulation's virtual image. Nevertheless, when considering the benefits afforded by POS for sharing, reusing, and maintaining simulation objects, the advantages outweigh its costs. With the techniques we are developing for optimized object access, POS will become even more attractive in the future.

2 PSE Objectives

The goal of PSE is to streamline the access of objects by "object prefetching". During the execution of a typical POS, objects are retrieved from secondary storage when required by the application program. Object replacement algorithms similar to those used for virtual memory, such as, "least recently used" are generally employed for swapping objects in and out of memory. Our methodology promotes a "supply-driven" model of object swapping rather than traditional "demand-driven" algorithms. A supply-driven methodology predicts in advance which objects the simulation will need and loads them into primary memory before the simulation requests them. However, to make predictions about the simulation's future data requirements, knowledge of the application and semantics of the simulation scenario is needed. Therefore, we categorize our work as "semantic-based object prefetching". Our techniques are based on the identification of a "working set" of objects for any active object being processed by the simulation. The working set consists of objects which have the potential to be subsequently requested. A working set can be defined by geographic locale, temporal locale, or

*This research was sponsored by the Air Force Human Resources Laboratory through the Defense Advanced Research Projects Agency under the auspices of RAND's National Defense Research Institute, a Federally Funded Research and Development Center sponsored by the Office of the Secretary of Defense. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official opinion of AFHRL, DARPA, the U.S. Government, or any person or agency connected with them.

semantic similarity with respect to the active object. Our initial testbed applications focus on a working set based on geographic locality.

2.1 Prefetching in PSE

Large-scale simulations, such as those done at RAND, may contain thousands of objects. Our laboratory has generated 80,000+ map objects for terrain based modeling. We find that up to 20,000 of these objects can be loaded into the CLOS environment on a workstation with 16 megabytes of main memory before the virtual memory system will have to do excessive paging to manage the size of the virtual image. Such excessive paging can greatly degrade the performance of the simulation. The persistent object capabilities allow objects to be prefetched based on the *geographic locality* of objects, instead of virtual address space locality. The geographic locality of map objects refers to objects that geographically reside near each other (eg. cities, roads, etc.). We claim that it is more efficient to swap objects based on geographic locality than virtual memory address locality, because as an object traverses the terrain, it is likely to interact with objects in nearby geographic locations. However, in a conventional virtual memory system there is no guarantee that objects with virtual address locality will be geographically local as well. The current prefetching capabilities of PSE fetch objects into main memory based on geographic locale.

Whether the number of objects is excessive or not, persistent objects reside on secondary memory. Objects can then be inspected and retrieved using the query language of the back-end object management system. The persistent objects can also be shared between simulations and can be reused. Our initial results indicate a fourfold speedup when reading 20,000+ previously formatted objects stored in our PSE file system, compared to reading and formatting the same objects each time for non-persistent CLOS.

3 PSE Architecture

We based the kernel implementation of PSE on the SOH (Shared Object Hierarchy which is part of the Picasso persistent object system) [Rowe 1986, Rowe 1988], obtained from UC Berkeley. We made the following modifications (described below) to Picasso to support our objectives: 1) made object identifiers remain constant for the life of the object; 2) added fields to the object handle; 3) reduced the size of the object directory by 50%; and 4) interfaced the Picasso shared object hierarchy with our own file system.

3.1 Object Identifiers

In Picasso a new object identifier is created each time an object is instantiated for a new application. In PSE, we have changed the use of object identifiers so that they remain constant for the life of the object. Thus, the object identifier is the representation of the pointer to an object in both primary and secondary memory. The object identifier works well as a symbolic pointer because to access a persistent object, the system uses the object identifier to hash into the persistent object directory which returns the object handle.

3.2 Object Handles

The object handle is a data structure which is stored in main memory and used to locate the object instance in the persistent object file. The object directory is a hash table that maps the object identifier into the object handle. An object handle contains the following fields:

1. *Id* – A unique id constant for the life of the object.
2. *Instance* – A pointer to a CLOS instance containing the object's attributes.
3. *Mode* – The update mode (*local-copy*, *deferred-update*, or *direct-update*).
4. *Indb* – Indicates if the object is stored in the database or not.
5. *Modified* – Indicates whether or not the instance has been modified.
6. *File-pointer* – A pointer to the object's location in secondary memory.
7. *Object-length* – The number of bytes for the object's representation in secondary memory.
8. *In-memory* – Indicates whether or not the object is resident in memory.

In our PSE implementation, we added the *file-pointer* and *object-length* fields to the object handle to implement the interface with our file system. We eliminated the *pinned* field, used in Picasso, because it was not necessary for our purposes.

3.3 Object Directory

In PSE, a directory maps an object's identifier to a directory "handle." When an object's identifier is referenced in the simulation, the object's handle is transparently accessed to determine the status of the object. For example, a flag in the handle indicates whether the object is resident in primary memory; another handle field records the object's location in secondary memory. Handles are invisible to the programmer, thereby achieving a seamless integration of the object system and the file management system. However, the object directory must permanently reside in primary memory because it is the only interface between PSE and the file system.

During POS processing, one major objective affecting performance is to maintain a large working set of objects in primary memory. We found that the directory for Picasso objects was especially large and occupied space that could otherwise be allocated to objects. Therefore, to accommodate the largest working set possible, we reduced the size of the PSE's directory by 50% through compaction techniques. Because these reduction techniques are independent of the file system interface, in the future we can port them with no change in their specification.

3.4 File-based Objects

The Picasso system interfaces with the extended relational DBMS, POSTGRES [Stonebraker and Rowe 1985]. In our implementation of PSE, we replaced POSTGRES with our own file system for more control over data manipulation and because our experimentation doesn't require the sophisticated DBMS facilities offered by POSTGRES.

The file system in PSE facilitates random access of objects. The file system inputs the object from secondary memory, using the file pointer, and instantiates it as a database object. We extended CLOS by defining methods to go indirectly through the handle's instance pointer to access and modify slots in an object. The ability to add methods to these low level primitives allows for a simple and transparent extension to CLOS that enhances it for persistence. PSE is loosely coupled to its file system, so when PSE becomes more mature, we can easily replace its current file manager with POSTGRES or some other DBMS.

4 Application Areas

We have acquired a large collection of map objects for terrain-based simulation. The map data is stored in a representation that can be converted into PSE objects. The map objects consist of roads, cities, road intersections, dead ends, rivers, and bridges. A class hierarchy to describe the map objects consists of edge objects (roads) and node objects (cities, road intersections, dead ends, and bridges). Since the map data is broken into nodes and edges, it applies well to graph theory algorithms. The most useful type of graph theory algorithm for the map data are shortest path algorithms. The shortest distance algorithm is used by a moving object when navigating a road network.

4.1 Dijkstra's Shortest Path Algorithm

To determine the shortest path, we have modified Dijkstra's shortest path algorithm [Gould 1988] to save the nodes traversed for the shortest path. Dijkstra's algorithm works by iteratively finding the *minimum label* from any of the nodes whose distance to the start node is known. It updates paths as it finds newer ones which are shorter. It then adds the closest node to its list of known nodes. To find the minimum label, Dijkstra's algorithm accesses all nodes adjacent to those whose distance from the start node is known. It must access the location fields in all these adjacent nodes to determine which node is nearest.

Prefetching of objects applies to the operation of iteratively finding the minimum label. While the algorithm is determining the distance to the current adjacent node, we can activate a separate process to retrieve the next adjacent node. The procedure continues until the minimum label has been found.

We have implemented a testbed for prefetching with the modified Dijkstra's algorithm. Using the multi-process facility of Allegro Common Lisp, we have written the code in manner similar to that used for a multiprocessor Lisp system. We have used Dijkstra's algorithm to monitor system parameters (described in section 5) to help determine the amount of speedup we can attain. Nonetheless, factors such as multiprocessor system overhead can not be determined on a uniprocessor system, and therefore we would like to eventually test prefetching techniques on a multiprocessor Lisp system to determine actual performance improvements.

4.2 Traveling Salesman Algorithm

An alternative file structure that we will be exploring in the future would allow for a different kind of prefetching than described for Dijkstra's algorithm. Because most geographic objects include a location attribute, these objects can be stored in a binary—tree file structure sorted on two dimensions: latitude and longitude. Persistent objects would reside in subtrees where all other objects in the subtree would have the same geographic proximity. As the simulation progresses, when an object not in memory is accessed, the system will retrieve the object and initiate a processor to retrieve all the objects geographically local to it. Such a methodology is similar to paging in a virtual memory system, except that the page is based on geographic locale instead of virtual memory addresses. Likewise, when a page of local objects is retrieved, if the number of objects in memory will infringe on performance, the system frees up space by writing out a page of objects. The system keeps a count each time an object is referenced and writes out pages of objects that are least recently used. Effectively, this constitutes a virtual memory system based on objects and geographic locale.

This alternative approach for fetching objects would apply better to the traveling salesman problem than to Dijkstra's shortest path algorithm. Dijkstra's algorithm does not ensure that all nodes accessed to find the minimum label will be geographically local to one another. However, the traveling salesman nearest neighbor heuristic moves from location to location and looks to its immediate neighbors to find the shortest traversal through the graph.

5 Analysis of Prefetching During Map Traversal

Prefetching of persistent objects decouples the input and instantiation of an object from the behavior and processing of a simulation. The key to object prefetching is the ability to predict in advance when objects will be needed by the simulation. Toward this goal we have been collecting performance data during numerous PSE trials to compare times for simulation processing against times for object management, i.e., reading and instantiating objects. Our testbed scenario consisted of the traversal of a connected graph (digital cartographic data) containing 79 edges (roads) and 87 vertices (road intersections). PSE supports programmer control over the number of objects resident in main memory by setting the value of the global variable *memory-full*. When the number of objects instantiated in main memory equals the value of *memory-full*, subsequent reference to an object not in memory causes garbage collection of unused memory-based objects to make room for a newly referenced object.

5.1 Experimental Results

In our discussion below, we identify three factors which play a role in the potential improvements afforded by object prefetching: (1) object maintenance, (2) object faults, and (3) "prediction-to-access gap." Tests revealed that, on average, 97% of the total run time time is spent for object maintenance. Only the remaining 3% is consumed by the application program. Furthermore, only 4% of the object management time occurred inside the operating system's I/O routines. Therefore, I/O time is small, but the time spent by Lisp building data structures for the input and instantiation of instances is extremely large. Figure 1 illustrates total execution time for map traversals with varying values of *memory-full*. If PSE allowed all referenced objects to reside in main memory concurrently, (by setting *memory-full* greater than 166, where 166 is the total number of objects), the run times are very small. In Figure 2, we conducted the same tests but instead recorded only the processor time spent for object management. Comparing these graphs indicates that input and instantiation of persistent objects is a critical area where performance improvements could have the greatest positive results. Timings also show that it costs roughly 90 CPU milliseconds to fetch an instance from secondary storage, although the number will undoubtably change once PSE is interfaced with a DBMS.

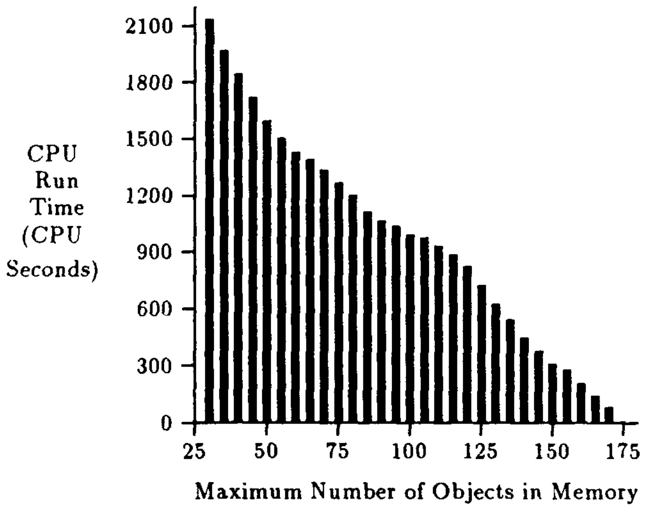

Figure 1: Run Times During Dijkstra Traversal

Another issue related to object maintenance which contributes to performance degradation is the occurrence of object faults. An object fault occurs when the simulation requests a slot value from an object that is not in primary

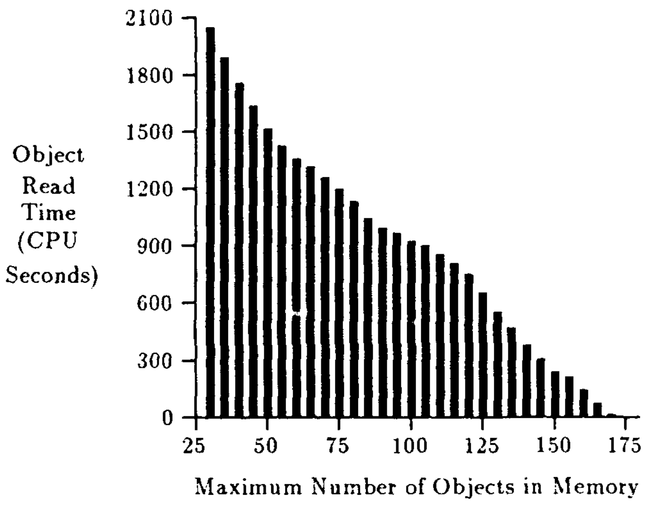

Figure 2: Object read times during Dijkstra traversal

memory which subsequently causes the system to read the instance from secondary memory, instantiate it, and return its value. By recording the frequency of object faults, we determined how often a simulation was being blocked by the need to access an object not resident in main memory. The graph in Figure 3 plotting object faults against differing values of *memory-full* shows the results which we expected.

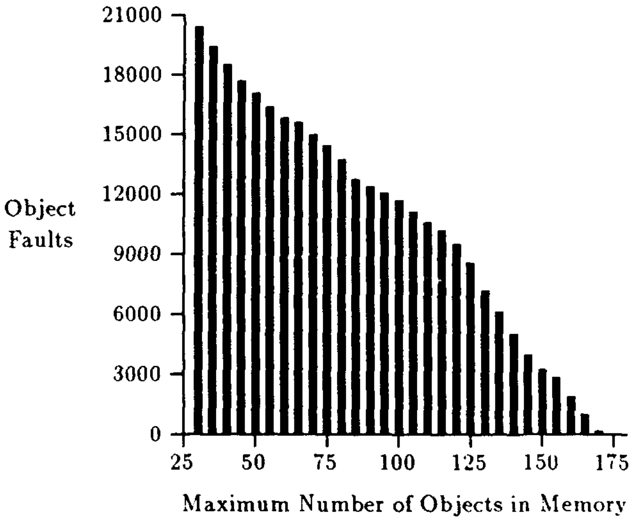

Figure 3: Object faults during Dijkstra traversal

Finally, the third factor which is important to consider, addresses the "prediction-to-access gap." This interval is the time between when the system predicts that it needs an object and when the object actually gets accessed. In order to effectively prefetch objects, two conditions are required: (1) the need to accurately predict some percentage of object faults and (2) evidence that, on average, the "prediction-to-access gap," is greater that the overhead for prefetching an object. Percentages of object faults which PSE can predict is shown is Figure 4. These results show that as the number of objects resident in memory increases, object prediction improves. One explanation for this phenomenon is that when the value of *memory-full* is small, memory fills up much quicker than when its value is large causing objects to be garbage collected more frequently. Thus, the likelihood that an object will be swapped out after it is prefetched, but before it is referenced, increases. As a result, prefetching becomes more than just the act of fetching an object before it is needed: the prefetching algorithm should be synchronized so that (1) the object is fetched before it is accessed and (2) the object is not swapped out of memory between the time when it has been fetched and the time when it will be referenced. This narrowing of the prediction-to-access gap results in a lower percentage of total object faults that can be predicted. The prefetching algorithm used for Figure 4, doesn't take into account that *memory-full* may be exceeded before an object is referenced, and thus predicts a lower amount of object faults as the value of *memory-full* is reduced. A smarter prefetching algorithm would be able to predict a larger percentage of object faults when *memory-full* is small and is an area worthy of further study.

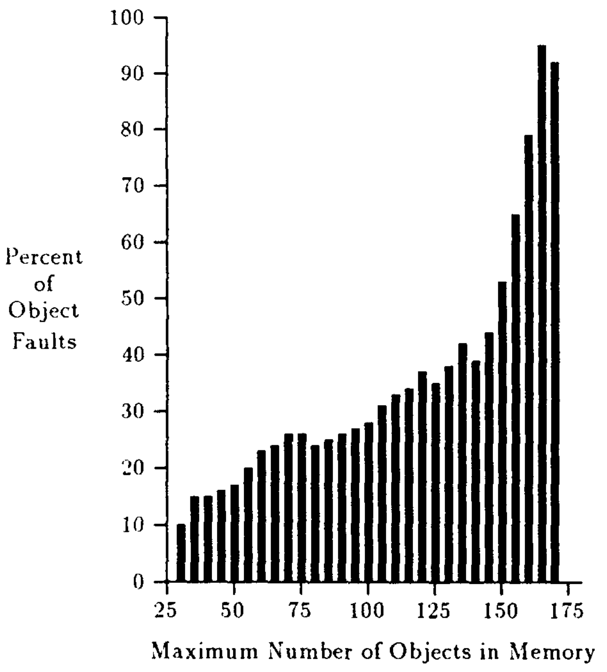

Figure 4: Percentage of total object faults which were predicted during Dijkstra traversal

5.2 Theoretical Results

Our initial findings suggest that prefetching may be a viable approach to improving performance under certain conditions. Figure 5 shows an abstracted time line of simulation execution distinguishing between application processing and object maintenance. In this diagram, the following time stamps and intervals are identified:

t_0	simulation begins
t_1	prediction that object O_1 will be referenced;
t_2	object O_1 is referenced; simulation is interrupted by object fault; fetching of object O_1 begins
t_3	fetching of object O_1 ends; simulation resumes
$t_2 - t_1$	prediction-to-access gap
$t_3 - t_2$	time to fetch object O_1; interval when simulation is blocked

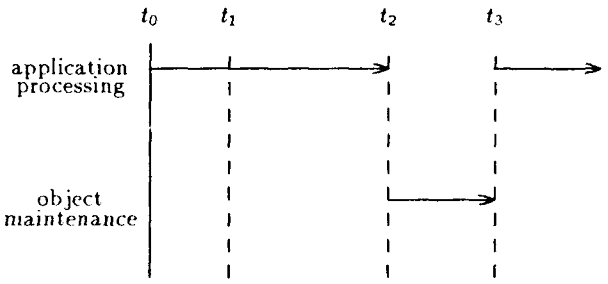

Figure 5: Time line for single processor execution

Since the costs of accessing and instantiating an instance from secondary storage are so high and have such a large impact on performance, it would be advantageous to interface object prefetching with a multiprocessor system. A multiprocessor PSE architecture will allow a separate processor to handle the input and instantiation of objects thus reducing the number of object faults. Three significant parameters which determine whether a multiprocessor system will result in performance improvements are (1) the time to prefetch an object; (2) the prediction-to-access gap; and (3) process overhead, that is, the overhead of spawning a prefetching process. Below, we present three cases based on different values of these parameters indicating what improvements could be expected with a multiprocessor PSE system. We also include a time line similar to Figure 5 for two of the three cases.

In the discussion below, let T_f, T_g, and T_o denote the prefetching time, prediction-to-access gap, and process overhead respectively.

Case 1: (See Figure 6)

If $T_o + T_f < T_g$

then the best performance improvement will be achieved; the simulation will perform as if all objects are memory resident.

The performance improvement of Case 1 over single processing is: $T_f - T_o$

In terms of Figure 6, the performance improvement is: $t_1'' - t_1'$ time units.

The following time stamps and intervals used in Figure 6	
t_0	simulation begins
t_1	prediction that object O_1 will be referenced; initialize prefetching processor
$t_1{\prime}$	prefetching of object O_1 begins
$t_1{\prime\prime}$	prefetching of object O_1 ends
t_2	object O_1 is referenced; simulation continues; no object fault because object was prefetched
$t_1{\prime} - t_1$	process overhead time $[T_o]$
$t_1{\prime\prime} - t_1{\prime}$	object prefetching time $[T_f]$
$t_2 - t_1$	prediction-to-access gap $[T_g]$

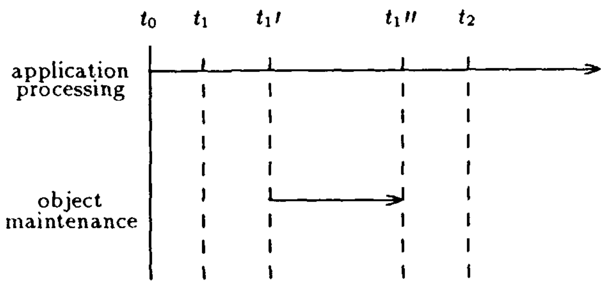

Figure 6: Time line for multiprocessor execution when prefetching terminates before object is referenced

The following time stamps and intervals used in Figure 7	
t_0	simulation begins
t_1	prediction that object O_1 will be referenced; initialize prefetching processor
$t_1{\prime}$	prefetching of object O_1 begins
t_2	object O_1 is referenced; simulation is interrupted by object fault
$t_1{\prime\prime}$	prefetching of object O_1 ends; simulation resumes
$t_1{\prime} - t_1$	process overhead time $[T_o]$
$t_2 - t_1$	prediction-to-access gap $[T_g]$
$t_1{\prime\prime} - t_1{\prime}$	object prefetching time $[T_f]$
$t_1{\prime\prime} - t_2$	interval when simulation is blocked

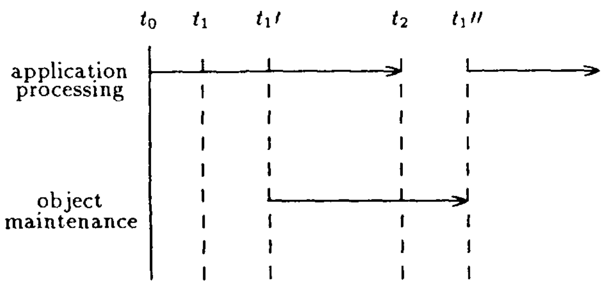

Figure 7: Time line for multiprocessor execution when prefetching terminates after object is referenced.

Case 2: (See Figure 7)

If $T_o < T_g$ *and* Case 1 does not hold
then performance improvement over single processing is:
$T_g - T_o$
and the simulation is blocked for:
$T_o + T_f - T_g$

In terms of Figure 7, the performance improvement is: $t_2 - t_1{\prime}$ *and* the interval when the simulation is blocked is: $t_1{\prime\prime} - t_2$ time units.

Case 3:

If $T_o >= T_g$
then multiprocessor prefetching will degrade performance compared to a single process system.

5.3 Converting Single Process Code to Enable Multiprocessor Prefetching

In the following section we present an example of how prefetching can be used in iterative loops where each pass through the loop accesses successive persistent objects in a list. Our experience in simulation programming shows that such iterative loops are a common operation. Loops similar to loop A below can be replaced by loops resembling B that utilize prefetching.

Loop A:

```
(DOLIST (OBJ OBJECT-LIST)
        (FOO OBJ))
```

Loop B:

```
(DO ((OBJ (CAR OBJECT-LIST) (CAR REST))
     (REST (CDR OBJECT-LIST) (CDR REST))
     (OBJ-TO-FETCH
      (WHEN (CDDR OBJECT-LIST)
            (PRE-FETCH (CADDR OBJECT-LIST)))
      (WHEN (CDDR REST)
            (PRE-FETCH (CADDR REST)))))
    ((NULL REST) NIL)
    (FOO OBJ))
```

Loop A sequentially accesses all of the objects in an object list and applies function FOO to each object. Loop B does the same, but it also prefetches an object during each iteration, beginning with the third object on through the end of the list. Experiments show that the prefetching in loop B predicted 88% of the total object faults caused by the loop.

6 Conclusions and Future Work

The conclusions we reached for performance improvements are strictly theoretical results; our future goal is to explore a multiprocessor architecture and compare the theoretical conclusions to empirical results. Toward this end, we are considering several multiprocessor Lisp systems [Goldman and Gabriel 1989, Zorn et. al. 1989, Halstead 1985] which are in various stages of development. One factor we must consider is how multiprocessor system overhead will effect performance, but it appears that the overhead is quite small. (Halstead mentions that Multilisp takes 6 milliseconds to create a future).

Another direction we will be seriously pursuing is to interface PSE with a commercial object-oriented DBMS, such as Gemstone, POSTGRES, Vbase, or Graphael. By using Gemstone as a backend object manager on a separate workstation, we may achieve some improvement by use of a dual-processor architecture over a single processor system. In parallel with investigating new architectures, we will also be experimenting with other prefetching techniques, other application programs and larger sets of memory and disk resident objects.

7 Acknowledgements

We'd like to thank Lawrence Rowe and Yongdong Wang for allowing access to the source code of the Shared Object Hierarchy.

References

[Atkinson and Buneman 1987] Atkinson, M. P., O. Peter Buneman, "Types and Persistence in Database Programming Languages", *ACM Computing Surveys*, Vol. 19, No. 2. June, 1987, pp. 105-190.

[Ford 1988] Ford, S., et. al., "Zeitgeist: Database Support for Object-Oriented Programming", *Proceedings of the 2nd International Workshop on Object-Oriented Database Systems*, Bad Munster am Stein-Ebernburg, FRG, September, 1988, pp. 23-42.

[Goldman and Gabriel 1989] Goldman, R. and Gabriel, R. P., "Qlisp: Parallel Processing in Lisp," *Proceedings of the 22nd Annual Hawaii International Conference on Systems Sciences*, January, 1989.

[Gould 1988] Gould, R. *Graph Theory*, Benjamin/Cummings, 1988.

[Halstead 1985] Halstead, R. H., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Languages and Systems October*, 1985.

[Khoshafian 1989] Khoshafian, S., "A Persistent Complex Object Database Language", *Data and Knowledge Engineering*, Vol. 3, 1989, pp. 225-243.

[Rowe 1986] Rowe, L. A., "A Shared Object Hierarchy," *Proceedings of the IEEE International Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, September 1986.

[Rowe 1988] Rowe, L. A., "PICASSO Shared Object Hierarchy+", *Proceedings of the First CLOS Users and Implementors Workshop*, Palo Alto, Ca, October 1988.

[Stonebraker and Rowe 1985] Stonebraker, M., Rowe L. A., "The Design of POSTGRES", Memorandum No. UCB/ERL 85/95, University of California, Berkeley, CA, November 15, 1985.

[Zorn et. al. 1989] Zorn, B., Ho K , Larus, J., Serenzato, L., and Hilfinger, P., "LISP Extensions for Multiprocessing," *Proceedings of the 22nd Annual Hawaii International Conference on Systems Sciences*, January, 1989.